

Local Reasoning with First-Class Heaps, and a New Frame Rule

Duc-Hiep Chu

National University of Singapore
hiepcd@comp.nus.edu.sg

Joxan Jaffar

National University of Singapore
joxan@comp.nus.edu.sg

Abstract

Separation Logic (SL) brought an advance to program verification of data structures by interpreting (recursively defined) predicates as *implicit* heaps, and using a separating conjoin operator to construct heaps from disjoint subheaps. While the Frame Rule of SL facilitated local reasoning in program fragments, its restriction to disjoint subheaps means that any form of sharing between predicates is problematic. With this as background motivation, we begin with an assertion language in which subheaps may be *explicitly* defined within predicates, and the effect of separation obtained by specifying that certain heaps are disjoint. The strength of this base language is not just its expressiveness, but it is amenable to *symbolic execution* and therefore automatic program verification. In this paper, we extend this base language with a new frame rule to accommodate subheaps and non-separating conjoining of subheaps so as to provide *compositional reasoning*. This significantly extends both the expressiveness and automatability of the base language. Finally we demonstrate our framework to automatically prove two significant example programs, one concerning a *summary* of a program fragments, and one exhibiting *structure sharing* in data structures.

1. Introduction

An important part of reasoning over heap manipulating programs is the ability to specify properties local to regions of memory. While traditional Hoare logic augmented with recursively defined predicates can be used (from as early as 1982 [Morris 1982; Bornat 2000]), it was Separation Logic [O’Hearn et al. 2001; Reynolds 2002] (SL) which made a significant advance. The key ideas here are: associating a predicate with a notion of *heap* and composing predicates

with the notion of *separating conjunction* of heaps, and accommodating *local reasoning* by means of a *frame rule*. A main shortcoming, however, is that the frame rule may not apply to a predicate which *shares* its heap with another (see [Hobor and Villard 2013] for a detailed discussion).

In this paper, we begin with an assertion language in which subheaps may be *explicitly* defined within predicates [Duck et al. 2013], and the effect of separation obtained by specifying that certain heaps are disjoint. In other words, heaps are *first-class* in this language. One main contribution of [Duck et al. 2013] is to refine the “overloaded meaning” of the separation conjunction, so that predicates can be conjoined in the traditional way.

In term of the assertion language, one advancement in this paper is that we remove the *implicit* “heap reality” of any subheaps appearing in a recursive predicate. Instead, heap reality is explicitly specified by connecting (ghost) subheaps to the distinguished heap variable \mathcal{M} , which represents the program global heap memory at the current state. We then show how to capture complex properties about *both* sharing and separation.

Our verification framework consists of two parts. First, we deal with the part of a heap that is *possibly changed* by a straight-line program fragment. This is handled by a *strongest postcondition* transform, so that the proof of a triple $\{ \phi \} P \{ \psi \}$ will just require the proof of ψ given the strongest postcondition of P from ϕ . This transformation, inherited from [Duck et al. 2013], can be easily automated, providing a basis towards automated verification.

The second and major part has to do with compositional reasoning and is about how to automatically frame properties of heap that is *definitely unchanged*. Indeed, the main contribution of this paper is a new frame rule to reclaim the power of compositional reasoning. Now why is the traditional frame rule from SL not adequate for our purpose?

A first reason is explained in [Duck et al. 2013]: that with a *strongest postcondition* approach to program verification, the frame rule, suitably translated into the language of explicit heaps, is simply not valid. In other words, if $\{ \phi \} P \{ \psi \}$ is established because ψ follows from the strongest postcondition of P executed from ϕ , it is not the

case that any heap separate from ϕ remains unchanged by the execution of P .

A second reason is that while the assertion language refers to multiple heaps, only those which are affected by the program must be isolated. In contrast, the traditional rule deals with a single (implicit) heap and so separation refers unambiguously to this heap alone.

Our new rule is used by explicitly naming *subheaps* in the specifications as part of the frame, in order to elegantly isolate relevant portions of the global heap \mathcal{M} . Consequently, a significant distinction is that our frame rule is concerned only on heap *updates*, as opposed to *all* heap references as in traditional SL. More specifically, we firstly facilitate the propagation of subheap properties from the precondition to the postcondition, when they are not involved in program heap updates. This is intuitively the key intention of a frame rule: the propagation of unaffected properties. Secondly and just as importantly, the rule needs also to propagate *separation* information. Toward this end, we introduce a concept of *evolution* in a triple: when a collection of subheaps in the precondition evolves to another collection of subheaps in the postcondition, it follows that separation from the first collection implies separation from the second. Thus while SL advanced Hoare reasoning with the implicit use of disjoint heaps, our logic advances SL with the explicit use of arbitrary subheaps.

Our frame rule provides for a verification framework to be *automatable*. Let us first be clear that the framework is used in “modular” program verification where procedures and loops are provided with specifications and/or invariants. This modularization provides natural abstraction boundaries that can be exploited in the reasoning process. (Further, each procedure or loop body can be analyzed just once, improving scalability.) Then, a standard process of analyzing the code, in conjunction with its specifications, generates a set of logical constraints called verification conditions (VC’s). Automation therefore is twofold: to come up with appropriate VC’s, and to have a *theorem-prover* to dispatch the VC’s. Now what we mean in this paper by “automatable” is that, for a very large class of applications:

- Our *assertion language is expressive enough*. This does not mean we generate the specifications automatically. Rather, it means that our assertion language can *encode* appropriate properties that are originated manually.
- The *state-of-the-art theorem-provers* suffice. This is not saying present theorem-provers can automatically prove all VC’s in our language, which is clearly impossible. Rather, we mean that our generated VC’s are *likely* to be provable. This arises from the fact that the assertion language allows us to encode just enough details so that the generated VC’s are, in some sense, at just the right level of detail.

Finally, in Section 6 we present *fully automated* proofs of two important case studies and argue that our verification framework is applicable to a large class of problems. The first case study is about *summarization* of a program fragment, where the essence is to reason about a heap *transformation*. The second case study deals with *structure sharing* in data structures. Here assertions comprise multiple recursive predicates that refer to some common heap locations and the challenge is to manage the commonality while at the same time exploit separation over certain subheaps that are limited to just few predicates.

2. Preliminaries

Separation Logic (SL) [Reynolds 2002] is a popular extension of Hoare Logic [Hoare 1969] for reasoning over *heap manipulating programs*. SL extends predicate calculus with new logical connectives (namely *empty heap* (**emp**), *singleton heap* ($p \mapsto v$), and *separating conjunction* ($F_1 * F_2$)) such that the structure of assertions reflects the structure of the underlying heap. For example, the precondition in the valid Separation Logic triple

$$\{ x \mapsto _ * y \mapsto 2 \} * x = *y + 1 \{ x \mapsto 3 * y \mapsto 2 \}$$

represents a heap comprised of two *disjoint singleton* heaps, indicating that both x and y are *allocated* and that location y points to the value 2. In the postcondition, x points to value 3, as expected. SL also allows *recursively-defined* heaps for reasoning over data structures, such as **list** and **tree**. An SL triple $\{ \phi \} P \{ \psi \}$ additionally guarantees that any state satisfying ϕ will not cause a memory access violation in P . For example, the triple $\{ \mathbf{emp} \} *x := 1 \{ x \mapsto 1 \}$ is *invalid* since x is a dangling pointer in any state satisfying the precondition.

We now overview the constraint language in [Duck et al. 2013]. We have a set of Values (e.g. integers) and we define Heaps to be all *finite partial maps* between values, i.e., $\text{Heaps} \stackrel{\text{def}}{=} (\text{Values} \rightarrow_{\text{fin}} \text{Values})$. There is a special value null (“null” pointer) and a special heap **emp** (“empty” heap). Where \mathcal{V}_v and \mathcal{V}_h denote the sets of value and heap variables respectively, our *heap expressions* HE are as follows:

$$\begin{aligned} H &::= \mathcal{V}_h & v &::= \mathcal{V}_v \\ HE &::= H \mid \mathbf{emp} \mid (v \mapsto v) \mid HE * HE \end{aligned}$$

An *interpretation* \mathcal{I} maps \mathcal{V}_h to Heaps and \mathcal{V}_v to Values. Syntactically, a *heap constraint* is a literal of the form $(HE \simeq HE)$. An interpretation \mathcal{I} satisfies a heap constraint $(HE_1 \simeq HE_2)$ iff $\mathcal{I}(HE_1) = \mathcal{I}(HE_2)$ are the same heap, and the separation properties within HE_1 and HE_2 hold.

Let $\text{dom}(H)$ be the *domain* of the heap H . As shown in [Duck et al. 2013], heap constraints can be normalized into three basic forms:

$$\begin{aligned} H &\simeq \mathbf{emp} && (\text{EMPTY}) \\ H &\simeq (p \mapsto v) && (\text{SINGLETON}) \\ H &\simeq H_1 * H_2 && (\text{SEPARATION}) \end{aligned}$$

where $H, H_1, H_2 \in \mathcal{V}_h$ and $p, v \in \mathcal{V}_v$. Here (EMPTY) constrains H to be the empty heap (i.e., $H = \emptyset$ as a set), (SINGLETON) constrains H to be the singleton heap mapping p to v (i.e., $H = \{(p, v)\}$ as sets), and (SEPARATION) constrains H to be the heap that is partitioned into two disjoint sub-heaps H_1 and H_2 (i.e., $H = H_1 \cup H_2$ as sets and $\text{dom}(H_1) \cap \text{dom}(H_2) = \emptyset$).

In addition to the basic heap constraints, we assume definitions for *sub-heap relation* ($H_1 \sqsubseteq H_2$), *domain membership* ($p \in \text{dom}(H)$), and (overloaded) for brevity, *separation relation* ($H_1 * H_2$). For example, writing $H_1 \sqsubseteq H_2$ is equivalent to $H_2 \simeq H_1 * _, p \in \text{dom}(H)$ to $H \simeq (p \mapsto _) * _, p \notin \text{dom}(H)$ to $_ \simeq H * (p \mapsto _)$, and $H_1 * H_2$ to $_ \simeq H_1 * H_2$; where the underscore in each instance denotes a fresh variable.

Finally, we have a *recursive constraint*. This is an expression of the form $p(h_1, \dots, h_n, v_1, \dots, v_m)$ where p is a user-defined *predicate symbol*, the $h_i \in \mathcal{V}_h, 0 \leq i \leq n$ and the $v_j \in \mathcal{V}_v, 0 \leq j \leq m$. Associated with such a predicate symbol is a *recursive definition*. We use the framework of *Constraint Logic Programming* (CLP) [Jaffar and Lassez 1987] to inherit its syntax, semantics, and its built-in notions of unfolding rules, for realizing recursive definitions. For brevity, we only informally explain the language. The following constitutes a recursive definition of $\text{list}(h, x)$, specifying a *skeleton list* in the heap h rooted at x .

$\text{list}(h, x) :- h \simeq \text{emp}, x = \text{null}.$
 $\text{list}(h, x) :- h \simeq (x \mapsto y) * h_1, \text{list}(h_1, y).$

Note that the comma-separated expressions in the body of each rule is either *value constraint* (e.g. $x = \text{null}$), a heap constraint (e.g. $h \simeq \text{emp}$), or a recursive constraint (e.g. $\text{list}(h_1, y)$). In the examples presented in later Sections, our value (i.e. “pure” or non-heap) constraints will either be arithmetic or basic set constraints over values.

The *semantics* of a set of rules is traditionally known as the “least model” semantics [Jaffar and Lassez 1987]. Essentially, this is the set of valuations of the expressions in the predicate that results in a *true* value when instantiated into the recursive rules. Thus for example, the semantics of the predicate $\text{list}(h, x)$ is simply that h is a heap which houses an acyclic list rooted at x and which is null terminated.

3. Programs and the Assertion Language

We assume a vanilla imperative programming language which has functions and recursive functions, but not loops. For the purpose of this paper, loops can always be compiled into tail-recursive functions. Other than standard non-heap statements, the language contains the following *heap manipulation statements*:¹

- sets x to be the value pointed to by y : $x = *y$;
- sets the value pointed to by x to be y : $*x = y$;

¹ Here we assume the (de)allocation of single heap cells. This can be easily generalized.

- points x to a freshly allocated cell: $x = \text{malloc}(1)$;
- deallocates the cell pointed to by x : $\text{free}(x)$.

Note that in the program syntax the heap is not explicitly mentioned. Instead, the heap is manipulated using the “*” notation as in the C language. (There should be no confusion with the separation operator “*” as understood in Separation Logic or in our heap constraint language.) Since our later discussion will involve symbolic execution, we also assume that branch condition is translated to *assume*($_$) statement.

Programs operate over an unbounded set of *program variables* \mathcal{V}_P , and these are the *value variables*. In other words, \mathcal{V}_P is a subset of \mathcal{V}_v . We use one distinguished heap variable \mathcal{M} to represent the *global heap memory*. Additional existential or *ghost* variables may appear in assertions. A ghost variable of type heap will be called a *subheap*.

The subheaps serve two essential and distinct purposes: (a) to describe subheaps of the global heap \mathcal{M} at the current program point, and (b) to describe some other “existential” heap. A common instance of (b) is the heap corresponding to the global heap at some *other* program point.

We use the terminology “ghost heap” in accordance to standard practice that subheaps are existential, but, in assertions, they can be used to constrain the value of the global heap. Importantly, as ghost variables, their values *cannot be changed* by the program. We will see later that this is important in practice because (a) predicates in assertions often need to be defined only using ghost subheaps, and (b) it is automatic that these predicates can be “framed through” any program fragment P because P cannot change the value of any ghost variable.

An *assertion* A is a formula over \mathcal{V}_P , \mathcal{M} , and ghost variables, and is of the form:

$$A ::= VF \mid HF \mid RC \mid A \wedge A \mid A \vee A$$

where VF is a value constraint, HF a heap constraint and RC a recursive constraint.

An *interpretation* of an assertion is obtained in the traditional way: value terms are interpreted into values, and heap terms are interpreted into finite maps from values to values, and finally value and heap formulas are interpreted into *false* or *true* (in which case it is a *model*).

Note that we shall present rules that define recursive constraints using fresh variables. Notationally, for heaps, we shall use the small letter ‘h’ in rules, while using the large letter \mathcal{H} in assertions. Also, we use “,” in assertions as shorthand for logical conjunction.

As an example, consider the annotated C-like program fragment and the rules defining the predicate `inc_list` below. The fragment increments all the data values in an acyclic list by 1.

```

struct node {
    int data;
    struct node *next;
};

{ list( $\mathcal{H}, x$ ),  $\mathcal{H} \sqsubseteq \mathcal{M}$  }
    y = x;
    while (y) {
        y->data++;
        y = y->next;
    }
{ inc_list( $\mathcal{H}_1, \mathcal{H}, x$ ),  $\mathcal{H}_1 \sqsubseteq \mathcal{M}$  }

inc_list( $h_1, h_2, x$ ) :-
     $h_1 \simeq \text{emp}, h_2 \simeq \text{emp}, x = \text{null}$ .
inc_list( $h_1, h_2, x$ ) :-
     $h_1 \simeq (x \mapsto (d+1, \text{next})) * h'_1$ ,
     $h_2 \simeq (x \mapsto (d, \text{next})) * h'_2$ ,
    inc_list( $h'_1, h'_2, \text{next}$ ).

```

The recursive constraint $\text{list}(\mathcal{H}, x)$, as before, describes a heap \mathcal{H} which houses an acyclic list rooted at x . The fact that this heap is part of the global heap memory is specified by the constraint $\mathcal{H} \sqsubseteq \mathcal{M}$. Unlike before, each node here has a data value in addition to a “next” pointer. Note that in a triple, a ghost variable appearing *both* in the precondition and postcondition means that we are referring to the same entity, e.g., \mathcal{H} above, while this is not the case for the program variable x or for the global heap memory \mathcal{M} . (By analogy, in Hoare Logic the triple $\{x = x_0\} x++ \{x = x_0 + 1\}$ describes an increment of x .)

The recursive constraint $\text{inc_list}(\mathcal{H}_1, \mathcal{H}, x)$ in the postcondition similarly defines that x is a list, but this time the list resides in the heap \mathcal{H}_1 . It has a second argument, the ghost heap \mathcal{H} , which importantly, also appears in the precondition. This allows us to conveniently state the (summary of) changes of subheaps done by the code. The definition of inc_list , shown above, not only states that each datum in the list rooted at x in \mathcal{H}_1 is one more than the corresponding datum in the list rooted at x in \mathcal{H} , but also states that all the links (the *next* pointers) are not modified.

It can now be seen that, in the postcondition, (a) the new heap of interest housing the list in the memory is \mathcal{H}_1 (by virtue of the constraint $\mathcal{H}_1 \sqsubseteq \mathcal{M}$), and (b) the values in this list are related to the list in the precondition because the latter list has been captured in the variable \mathcal{H} , which in the postcondition is a ghost variable. It is important to note also that the recursive constraint $\text{list}(\mathcal{H}, x)$ is unaffected by the code and still holds at the postcondition, though its *usefulness* is questionable. On the other hand, $\mathcal{H} \sqsubseteq \mathcal{M}$ no longer holds at the postcondition, because the memory \mathcal{M} has been updated.

4. Symbolic Execution with Explicit Heaps

Symbolic execution involves executing a program using *symbolic values* as inputs, and it can be used for program verification in a standard way, as follows. We start with a precondition. The output of symbolic execution on a program path is a formula obtained at the end of a path, or the *strongest postcondition* of the precondition. For a loop-free program with no function call, symbolic execution facili-

tates program verification by considering a disjunction of all such path postconditions, which must then imply the desired postcondition. With function calls (or loops), we often need the help from the frame rule.

We now describe how to obtain a strongest postcondition from a given precondition ϕ [Duck et al. 2013].

Proposition 1 (Strongest Postcondition). *In the following Hoare-triples, the postcondition shown is the strongest postcondition of the primitive heap operation with respect to a precondition ϕ .*

$\{\phi\} x = \text{malloc}(1) \{ \text{alloc}(\phi, x) \}$	(Heap allocation)
$\{\phi\} \text{free}(x) \{ \text{free}(\phi, x) \}$	(Heap deallocation)
$\{\phi\} x = *y \{ \text{access}(\phi, y, x) \}$	(Heap access)
$\{\phi\} *x = y \{ \text{assign}(\phi, x, y) \}$	(Heap assignment)

where the auxiliary macros `alloc`, `free`, `access`, and `assign` expand as follows:

$\text{alloc}(\phi, x)$	$\stackrel{\text{def}}{=} \mathcal{M} \simeq (x \mapsto v) * \mathcal{H} \wedge \phi[\mathcal{H}/\mathcal{M}, v_1/x]$
$\text{free}(\phi, x)$	$\stackrel{\text{def}}{=} \mathcal{H} \simeq (x \mapsto v) * \mathcal{M} \wedge \phi[\mathcal{H}/\mathcal{M}]$
$\text{access}(\phi, y, x)$	$\stackrel{\text{def}}{=} \mathcal{M} \simeq (y \mapsto x) * \mathcal{H} \wedge \phi[v/x]$
$\text{assign}(\phi, x, y)$	$\stackrel{\text{def}}{=} \mathcal{M} \simeq (x \mapsto y) * \mathcal{H}_1 \wedge \mathcal{H} \simeq (x \mapsto v) * \mathcal{H}_1 \wedge \phi[\mathcal{H}/\mathcal{M}]$

where \mathcal{H} and \mathcal{H}_1 are fresh heap variables, and v and v_1 are fresh value variables. The notation $\phi[x/y]$ means formula ϕ with variable x substituted for y . \square

```

{  $\mathcal{H}_{99} \simeq \mathcal{M}$  }
    t1 = *x;
{  $\mathcal{M} \simeq (x \mapsto t_1) * \mathcal{H}_1, \mathcal{H}_{99} \simeq \mathcal{M}$  }
    *x = t1 + 1;
{  $\mathcal{M} \simeq (x \mapsto t_1 + 1) * \mathcal{H}_1, \mathcal{H}_2 \simeq (x \mapsto t_1) * \mathcal{H}_1, \mathcal{H}_2 \simeq (x \mapsto t_1) * \mathcal{H}_1, \mathcal{H}_{99} \simeq \mathcal{H}_2$  }
     $\Downarrow$  // (simplification)
{  $\mathcal{M} \simeq (x \mapsto t_1 + 1) * \mathcal{H}_1, \mathcal{H}_{99} \simeq (x \mapsto t_1) * \mathcal{H}_1$  }
    t2 = *x;
{  $\mathcal{M} \simeq (x \mapsto t_2) * \mathcal{H}_3, \mathcal{M} \simeq (x \mapsto t_1 + 1) * \mathcal{H}_1, \mathcal{H}_{99} \simeq (x \mapsto t_1) * \mathcal{H}_1$  }
    *x = t2 - 1;
{  $\mathcal{M} \simeq (x \mapsto t_2 - 1) * \mathcal{H}_4, \mathcal{H}_5 \simeq (x \mapsto v) * \mathcal{H}_4, \mathcal{H}_5 \simeq (x \mapsto t_2) * \mathcal{H}_3, \mathcal{H}_5 \simeq (x \mapsto t_1 + 1) * \mathcal{H}_1, \mathcal{H}_{99} \simeq (x \mapsto t_1) * \mathcal{H}_1$  }

```

Figure 1: Demonstrating Symbolic Execution

Let us now present a simple example, which should demonstrate the usefulness (and partly the correctness) of Proposition 1. Consider proving:

$$\{ \mathcal{H}_{99} \simeq \mathcal{M} \} *x += 1; *x -= 1; \{ \mathcal{H}_{99} \simeq \mathcal{M} \},$$

that is, to prove that the heap is unchanged after an increment and then a decrement of the cell x . In order to use Proposition 1 conveniently, we rewrite the program so that only one heap operation is performed per program statement; in Fig. 1 we show the rewritten program fragment together with the propagation of the formulas. (For brevity, we also perform a simplification step.) It is then easy to show that the

final formula implies $\mathcal{H}_{99} \simeq \mathcal{M}$, by first establishing that $\mathcal{H}_1 \simeq \mathcal{H}_3 \simeq \mathcal{H}_4$ and $v = t_2 = t_1 + 1$. Note that this example provides a *summary* of the program, a mapping from input to output variables (and in this case, that the heap is the same before and after execution).

5. The Frame Rule

The frame rule is the key towards having modular verification. In traditional Hoare logic, an assertion, which does not mention heap variables, can be framed through a program fragment if the fragment does not modify any (free) variable in the assertion.

Proposition 2 (Classic Frame Rule).

$$\frac{\{\phi\} P \{\psi\}}{\{\phi \wedge \pi\} P \{\psi \wedge \pi\}} \text{Mod}(P) \cap FV(\pi) = \emptyset \quad (\text{CFR})$$

where $\text{Mod}(P)$ denotes the variables that P modifies, and $FV(\pi)$ denotes the free variables of π . \square

In the setting of our assertion language, this kind of framing takes place for the predicate π whose free heap variables are ghost, since these variables cannot be modified by P . We will see later that not all properties of interest can be framed. In fact, since the global heap memory can in general be changed by P , what *cannot* be framed through is the property that a ghost variable \mathcal{H} is consistent with the global heap memory \mathcal{M} , i.e., $\mathcal{H} \sqsubseteq \mathcal{M}$. We call such a property the “heap reality” of \mathcal{H} .

In Separation Logic, where the heap is of the main interest, a key step is that when a program fragment is “enclosed” in some heap, then any formula π whose “footprint” is separate from this heap can be framed through the program. In the frame rule of SL:

$$\frac{\{\phi\} P \{\psi\}}{\{\phi * \pi\} P \{\psi * \pi\}}$$

the premise $\{\phi\} P \{\psi\}$ ensures that the implicit heap arising from the formula ϕ captures all the heap accesses, read or write, in the program fragment P . (Note that, as before, we do need to satisfy the standard side condition of the frame rule that the intersection, of the set of variables changed by P and the set of free variables in π , is empty. However, for brevity, we choose not to focus on this side condition when we discuss the frame rule in the rest of this Section.) We now can extend the proof in the premise to that in the conclusion because π can be framed by virtue of it being on a heap separate from ϕ .

In our setting, the concept of enclosure we use is roughly to have an explicit subheap (or a collection of subheaps) which contains the program heap updates. These updates are defined to be the cells that the program *writes to*, or *deallocates*. This is because the property $\mathcal{H} \sqsubseteq \mathcal{M}$, where \mathcal{H} is a ghost variable, is falsified *just in case* the program code has written to or deallocated some cell in \mathcal{M} whose address is also in $\text{dom}(\mathcal{H})$. In other words, the heap reality

of \mathcal{H} is lost. Note that **malloc** changes \mathcal{M} , but it does not affect cells that have already been part of \mathcal{M} .

Definition 1 (Heap Update). *Given an address value v , a heap update to location v is defined as a statement that either writes to or deallocates the location v .*

Before formalizing our notion of “enclosure”, however, we first need a concept of heap “evolution”. Let us use the notation $\tilde{\mathcal{H}}$ to denote the union $\bigcup_i \mathcal{H}_i$ of a collection of subheaps $\mathcal{H}_1, \dots, \mathcal{H}_n$, $n \geq 2$. Thus for example, $\tilde{\mathcal{H}} \sqsubseteq \mathcal{M}$ simply abbreviates $\mathcal{H}_1 \sqsubseteq \mathcal{M} \wedge \dots \wedge \mathcal{H}_n \sqsubseteq \mathcal{M}$.

Definition 2 (Evolution). *Given a valid triple $\{\phi\} P \{\psi\}$, we say that a collection $\tilde{\mathcal{H}}$ in ϕ , where $\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M}$, evolves to a collection $\tilde{\mathcal{H}}'$ in ψ , where $\psi \models \tilde{\mathcal{H}}' \sqsubseteq \mathcal{M}$, if for each model \mathcal{I} of ϕ , executing P from \mathcal{I} will result in \mathcal{I}' , such that for any (address) value v , $v \in (\text{dom}(\mathcal{I}(\mathcal{M})) - \text{dom}(\mathcal{I}(\tilde{\mathcal{H}})))$ implies $v \notin \text{dom}(\mathcal{I}'(\tilde{\mathcal{H}}'))$.*

We shall use the notation $\{\phi\} P \{\psi\} \rightsquigarrow \text{EVOLVE}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')$ to denote such evolution. \square

Intuitively, $\{\phi\} P \{\psi\} \rightsquigarrow \text{EVOLVE}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')$ means that the largest \mathcal{H}' can be is \mathcal{H} plus any new cells allocated by P , and minus any that are freed by P . Note also that because the triple is valid, \mathcal{I}' will be a model of ψ . One important usage of the evolution concept is as follows: any heap \mathcal{H}_i such that $\mathcal{H}_i * \tilde{\mathcal{H}}$ and $\mathcal{H}_i \sqsubseteq \mathcal{M}$ at the point of the precondition ϕ (i.e., before P is executed), \mathcal{H}_i will be separate from $\tilde{\mathcal{H}}'$ at the point of the postcondition (i.e., after P is executed)².

Consider the **struct** node (for list) defined in Section 3 and the triple shown below.

$$\begin{aligned} & \{ \text{list}(\mathcal{H}_1, x), \mathcal{H}_1 \sqsubseteq \mathcal{M} \} \\ & \quad z = \text{malloc}(\text{sizeof}(\text{struct node})); \\ & \quad z \rightarrow \text{next} = x; \\ & \{ \text{list}(\mathcal{H}'_1, z), \mathcal{H}'_1 \sqsubseteq \mathcal{M} \} \end{aligned}$$

We say that \mathcal{H}'_1 is an *evolution* of \mathcal{H}_1 , or notationally, $\text{EVOLVE}(\mathcal{H}_1, \mathcal{H}'_1)$. Now assume that the triple represents only a local proof (i.e., we are also interested in other parts of \mathcal{M}). How should we compose this local triple to obtain a new triple? Formally, we have the following:

$$\frac{\{\phi\} P \{\psi\} \rightsquigarrow \text{EVOLVE}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')}{\{\phi \wedge \tilde{\mathcal{H}} * \mathcal{H}_0 \wedge \mathcal{H}_0 \sqsubseteq \mathcal{M}\} P \{\psi \wedge \tilde{\mathcal{H}}' * \mathcal{H}_0\}} \quad (\text{EV})$$

Theorem 1 (Propagation of Separation). *The rule (EV) is correct. \square*

Proof. (Sketch.) Let \mathcal{I} be a model of ϕ that is also a model of $\mathcal{H} * \mathcal{H}_0 \wedge \mathcal{H}_0 \sqsubseteq \mathcal{M}$. Let \mathcal{I}' be the result of executing P from \mathcal{I} . For each address $v \in \text{dom}(\mathcal{I}'(\mathcal{H}_0))$, because \mathcal{H}_0 is a ghost variable, i.e., its domain is not affected by executing P , we also have $v \in \text{dom}(\mathcal{I}(\mathcal{H}_0))$. It follows that $v \in (\text{dom}(\mathcal{I}(\mathcal{M})) - \text{dom}(\mathcal{I}(\tilde{\mathcal{H}})))$. Directly from the

² The condition $\mathcal{H}_i \sqsubseteq \mathcal{M}$ is to cater for the assumption that **malloc** can reuse memory cells which have been freed and reclaimed. If it is not the case, this condition can be dropped.

definition of evolution, we deduce $v \notin \text{dom}(\mathcal{I}'(\tilde{\mathcal{H}}'))$ must hold. As a result, \mathcal{I}' also satisfies $\tilde{\mathcal{H}}' * \mathcal{H}_0$. \square

We are now ready to describe our notion of enclosure. We wish to describe, given a program P and a heap collection $\tilde{\mathcal{H}}$ in a precondition description ϕ , that all heap updates (heap assignments or deallocations) in P , are confined to an evolution of $\tilde{\mathcal{H}}$. The following definition, intuitively, is about one (but not all) aspect of memory-safety.

Definition 3 (Enclose). *Suppose we have a valid triple $T = \{\phi\} P \{-\}$, $\tilde{\mathcal{H}}$ appears in ϕ , and that $\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M}$. We say $\tilde{\mathcal{H}}$ encloses all heap updates of P if for any model \mathcal{I} of ϕ and for any execution path of P of the form $P_1; s; P_2$ where s is a heap update to a location v , it follows that there exists $\tilde{\mathcal{H}}'$ s.t. $\{\phi\} P_1 \{-\} \rightsquigarrow \text{EVOLVE}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')$ and $v \in \text{dom}(\mathcal{I}'(\tilde{\mathcal{H}}'))$ hold, where \mathcal{I}' is the result of executing P_1 from \mathcal{I} .*

We will use the notation $T \rightsquigarrow \text{ENCLOSE}(\tilde{\mathcal{H}})$ to denote that $\tilde{\mathcal{H}}$ encloses all the updates of P wrt. T . \square

We can now introduce our frame rule. It is in fact all about “preserving the heap reality”. Recall that a recursive constraint, which satisfies the standard side condition and of which the heap variables $\tilde{\mathcal{H}}$ are all ghost (and this is a common situation), *remains true* from precondition to postcondition. What may no longer hold in the postcondition is the heap reality of $\tilde{\mathcal{H}}$. That is, $\tilde{\mathcal{H}} \sqsubseteq \mathcal{M}$ may hold at the precondition, but no longer so at the postcondition. In other words, given local reasoning for a code fragment P and the fact that $\mathcal{H} \sqsubseteq \mathcal{M}$ holds before executing P , how would we preserve this heap reality, without the need to reconsider the code fragment P ? Our answer is the following Hoare-style rule, our new frame rule:

$$\frac{\{\phi\} P \{\psi\} \rightsquigarrow \text{ENCLOSE}(\tilde{\mathcal{H}})}{\{\phi \wedge \tilde{\mathcal{H}} * \mathcal{H}_0 \wedge \mathcal{H}_0 \sqsubseteq \mathcal{M}\} P \{\psi \wedge \mathcal{H}_0 \sqsubseteq \mathcal{M}\}} \quad (\text{FR})$$

Theorem 2 (Frame Rule). *The rule (FR) is correct.* \square

Proof. (Sketch.) We prove by contradiction. Assume it is not the case, meaning that there is model \mathcal{I} of ϕ that is also a model of $\tilde{\mathcal{H}} * \mathcal{H}_0 \wedge \mathcal{H}_0 \sqsubseteq \mathcal{M}$ and \mathcal{I}' is the result of executing P from \mathcal{I} , but \mathcal{I}' does not satisfy $\mathcal{H}_0 \sqsubseteq \mathcal{M}$. Thus there must be a cell $(v \mapsto _)$ that belongs to $\mathcal{I}'(\mathcal{H}_0)$ but not $\mathcal{I}'(\mathcal{M})$. Because $\mathcal{I}(\mathcal{H}_0) \sqsubseteq \mathcal{I}(\mathcal{M})$, the fragment P must have updated the location v . Therefore, there must be an execution path of P which is of the form $P_1; s; P_2$, where s is a heap update to the location v . Let $\tilde{\mathcal{I}}$ be the result of executing P_1 from \mathcal{I} . By the definition of enclosure, assume $\{\phi\} P_1 \{-\} \rightsquigarrow \text{EVOLVE}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')$ and $v \in \text{dom}(\tilde{\mathcal{I}}(\tilde{\mathcal{H}}'))$ hold. By (EV) rule, we have $\tilde{\mathcal{I}}$ satisfies $\tilde{\mathcal{H}}' * \mathcal{H}_0$. Since \mathcal{H}_0 is a ghost variable, its domain is not affected by executing P_1 , i.e., $v \in \text{dom}(\tilde{\mathcal{I}}(\mathcal{H}_0))$ holds. This is a contradiction. \square

Let us demonstrate the use of the two theorems on a very simple example. Consider the triple:

$$\begin{aligned} & \{ ((x \mapsto _) * \mathcal{H}) \sqsubseteq \mathcal{M} \} \\ & \quad *x = 1; \\ & \{ ((x \mapsto 1) * \mathcal{H}) \sqsubseteq \mathcal{M} \} \end{aligned}$$

Of course, we could follow the symbolic execution rule presented in Section 4 and also be able to prove this triple. But, for the sake of discussion, we consider local reasoning on the following triple T :

$$\{ (x \mapsto _) \sqsubseteq \mathcal{M} \} * x = 1; \{ (x \mapsto 1) \sqsubseteq \mathcal{M} \},$$

which holds trivially. Also, we can clearly see that both $T \rightsquigarrow \text{EVOLVE}((x \mapsto _), (x \mapsto 1))$ and $T \rightsquigarrow \text{ENCLOSE}((x \mapsto _))$ hold. Applying the rule (EV), we deduce that $(x \mapsto 1) * \mathcal{H}$ holds after executing the code fragment. On the other hand, applying the frame rule, rule (FR), we deduce that $\mathcal{H} \sqsubseteq \mathcal{M}$ remains true, i.e., the heap reality of \mathcal{H} is preserved. Putting the pieces together, we can establish the truth of the original triple by making use of the two theorems.

Note that we only have (normal) conjunction, as opposed to separating conjunction in SL. So all the rules presented above can be used in combination. It is worthwhile to recall the following rule in classic Hoare logic:

$$\frac{\{\phi\} P \{\psi_1\} \quad \{\phi\} P \{\psi_2\}}{\{\phi\} P \{\psi_1 \wedge \psi_2\}}$$

We now revisit the `list` example presented earlier in this Section, but in the context that there exists a tree housed by a heap \mathcal{H}_2 that is separate from \mathcal{H}_1 . Consider the triple T below, and let P stand for the program fragment of interest. From the fact that $T \rightsquigarrow \text{EVOLVE}(\mathcal{H}_1, \mathcal{H}'_1)$ and $T \rightsquigarrow \text{ENCLOSE}(\text{emp})$ hold, and by applying the two rules (EV) and (FR), we can easily establish the validity of this triple.

$$\begin{aligned} & \{ \text{list}(\mathcal{H}_1, x), \text{tree}(\mathcal{H}_2, y), \mathcal{H}_1 * \mathcal{H}_2 \sqsubseteq \mathcal{M} \} \\ & \quad z = \text{malloc}(\text{sizeof}(\text{struct node})); \\ & \quad *z = x; \\ & \{ \text{list}(\mathcal{H}'_1, z), \text{tree}(\mathcal{H}_2, y), \mathcal{H}'_1 * \mathcal{H}_2 \sqsubseteq \mathcal{M} \} \end{aligned}$$

We next elaborate the connection of our two rules (EV) and (FR) with the traditional frame rule in Separation Logic (SL). First, why do we have two rules while SL has one, as introduced in the beginning of this Section? The reason is that SL, succinctly, captures *two* important properties: that

- π can be added to precondition ϕ and it *remains true* in the postcondition;
- π *retains its separateness*, from precondition ϕ to postcondition ψ .

The second property is important for *successive* uses of the frame rules. Our rule (FR) above only provides for the first property. We accommodate the second property with the other rule (EV), i.e., the “propagation of separation” rule.

The two concepts of evolution and enclosure in fact exist in SL, *implicitly*. Given the triple $T = \{\phi\} P \{\psi\}$, assume that \mathcal{H} is the heap housing the precondition ϕ and

$\frac{\text{[MALLOC]}}{\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M} \quad \psi \models \tilde{\mathcal{H}}' \sqsubseteq \mathcal{M} \quad \text{dom}(\tilde{\mathcal{H}}') \subseteq \text{dom}(\tilde{\mathcal{H}}) \cup \{x\} \quad \{ \phi \} x = \text{malloc}(1) \{ \psi \} \rightsquigarrow \text{EVOLVE}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')}$
$\frac{\text{[FREE]}}{\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M} \quad \psi \models \tilde{\mathcal{H}}' \sqsubseteq \mathcal{M} \quad \text{dom}(\tilde{\mathcal{H}}') \subseteq \text{dom}(\tilde{\mathcal{H}}) \setminus \{x\} \quad \{ \phi \} \text{free}(x) \{ \psi \} \rightsquigarrow \text{EVOLVE}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')}$
$\frac{\text{[OTHER-STATEMENTS]}}{\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M} \quad \psi \models \tilde{\mathcal{H}}' \sqsubseteq \mathcal{M} \quad \text{dom}(\tilde{\mathcal{H}}') \subseteq \text{dom}(\tilde{\mathcal{H}}) \quad \{ \phi \} s \{ \psi \} \rightsquigarrow \text{EVOLVE}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')}$
$\frac{\text{[SEQ-COMPOSITION]}}{\frac{\{ \phi \} P \{ \psi \} \rightsquigarrow \text{EVOLVE}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}') \quad \{ \psi \} Q \{ \gamma \} \rightsquigarrow \text{EVOLVE}(\tilde{\mathcal{H}}', \tilde{\mathcal{H}}'')}{\{ \phi \} P; Q \{ \gamma \} \rightsquigarrow \text{EVOLVE}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')}}}$
$\frac{\text{[CALL]}}{[\{ \phi \} \mathbf{f}() \{ \psi \} \rightsquigarrow \text{EVOLVE}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')] \in \text{Specs} \quad \phi' \models \phi \quad \{ \phi' \} \text{call } \mathbf{f}() \{ - \} \rightsquigarrow \text{EVOLVE}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')}$
$\frac{\text{[COMPOSITION]}}{\frac{\{ \phi \} P \{ \psi \} \rightsquigarrow \text{EVOLVE}(\tilde{\mathcal{H}}_1, \tilde{\mathcal{H}}'_1) \quad \phi \models \tilde{\mathcal{H}}_2 \sqsubseteq \mathcal{M} \quad \{ \phi \} P \{ \psi \} \rightsquigarrow \text{ENCLOSE}(\tilde{\mathcal{H}}) \quad \tilde{\mathcal{H}} * \tilde{\mathcal{H}}_2}{\{ \phi \} P \{ \psi \} \rightsquigarrow \text{EVOLVE}(\tilde{\mathcal{H}}_1 \cup \tilde{\mathcal{H}}_2, \tilde{\mathcal{H}}'_1 \cup \tilde{\mathcal{H}}_2)}}$

Figure 2: Hoare-style Rules for Evolution

\mathcal{H}' is the heap housing the postcondition ψ . In SL, the frame rule also requires that $T \rightsquigarrow \text{EVOLVE}(\mathcal{H}, \mathcal{H}')$ and that $T \rightsquigarrow \text{ENCLOSE}(\mathcal{H})$. Because our assertion language allows for the usage of multiple subheaps, which entails more expressive power, but we no longer can resort to such a default. For the purpose of this paper, we require the specifications to also nominate the subheaps participating in the evolution and/or enclosure relations. Such relations are stated under the keyword **frame**, following the typical **requires** and **ensures** keywords. We will demonstrate this when we present our driving examples in Section 6.

The next question of interest is how the evolution and enclosure relations are practically checked. For evolution, we use the rules in Figure 2. In the rule [CALL],

$$[\{ \phi \} \mathbf{f}() \{ \psi \} \rightsquigarrow \text{EVOLVE}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')] \in \text{Specs}$$

means that we have nominated $\text{EVOLVE}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')$ the specifications of function \mathbf{f} . Similarly for enclosure relation, which can be effectively checked using the rules presented in Figure 3. We further note that the checking of evolution and enclosure relations is also performed modularly. Specifically, at call sites, we make use of the rule [CALL] and then achieve compositional reasoning with the rule [COMPOSITION].

We finally conclude this Section with two Lemmas about the correctness of the rules presented in Figure 2 and Figure 3. For space reason, and because the rules are relatively intuitive, we omit the proofs.

$\frac{\text{[HEAP-ASSIGN]}}{\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M} \quad x \in \text{dom}(\tilde{\mathcal{H}}) \quad \{ \phi \} *x = y \{ - \} \rightsquigarrow \text{ENCLOSE}(\tilde{\mathcal{H}})}$
$\frac{\text{[FREE]}}{\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M} \quad x \in \text{dom}(\tilde{\mathcal{H}}) \quad \{ \phi \} \text{free}(x) \{ - \} \rightsquigarrow \text{ENCLOSE}(\tilde{\mathcal{H}})}$
$\frac{\text{[OTHER-STATEMENTS]}}{\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M} \quad \{ \phi \} s \{ - \} \rightsquigarrow \text{ENCLOSE}(\tilde{\mathcal{H}})}$
$\frac{\text{[SEQ-COMPOSITION]}}{\frac{\{ \phi \} P \{ \psi \} \rightsquigarrow \text{ENCLOSE}(\tilde{\mathcal{H}}) \quad \{ \phi \} P \{ \psi \} \rightsquigarrow \text{EVOLVE}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}') \quad \{ \psi \} Q \{ \gamma \} \rightsquigarrow \text{ENCLOSE}(\tilde{\mathcal{H}}')}{\{ \phi \} P; Q \{ \gamma \} \rightsquigarrow \text{ENCLOSE}(\tilde{\mathcal{H}})}}$
$\frac{\text{[CALL]}}{[\{ \phi \} \mathbf{f}() \{ \psi \} \rightsquigarrow \text{ENCLOSE}(\tilde{\mathcal{H}})] \in \text{Specs} \quad \phi' \models \phi \quad \{ \phi' \} \text{call } \mathbf{f}() \{ - \} \rightsquigarrow \text{ENCLOSE}(\tilde{\mathcal{H}})}$
$\frac{\text{[COMPOSITION]}}{\frac{\{ \phi \} P \{ \psi \} \rightsquigarrow \text{ENCLOSE}(\tilde{\mathcal{H}}) \quad \phi \models \tilde{\mathcal{H}}' \sqsubseteq \mathcal{M}}{\{ \phi \} P \{ \psi \} \rightsquigarrow \text{ENCLOSE}(\tilde{\mathcal{H}} \cup \tilde{\mathcal{H}}')}}}$

Figure 3: Hoare-style Rules for Update Enclosure

Lemma 1 (Evolution). *Given a valid triple $T = \{ \phi \} P \{ \psi \}$ where $\phi \models \tilde{\mathcal{H}}$ and $\psi \models \tilde{\mathcal{H}}'$, $T \rightsquigarrow \text{EVOLVE}(\tilde{\mathcal{H}}, \tilde{\mathcal{H}}')$ holds if it follows from the rules in Figure 2. \square*

Lemma 2 (Enclose). *Given a valid triple $T = \{ \phi \} P \{ - \}$ where $\phi \models \tilde{\mathcal{H}}$, $T \rightsquigarrow \text{ENCLOSE}(\tilde{\mathcal{H}})$ holds if it follows from the rules in Figure 3. \square*

6. Driving Examples

We have implemented a prototype. The main elements of the implementation are clearly the symbolic execution algorithm and the frame rule whose assembly into a total package is assumed to be well-understood:

- We assume that function specifications³ are given.
- For each function of interest, we consider the proof for one symbolic path at a time. For basic program statements, symbolic execution rules presented in Proposition 1 are used.
- At function calls, the frame rules in Section 5 are employed to achieve compositional reasoning.

With respect to automation, the remaining challenge, but outside the scope of this paper, is how to discharge the proofs of entailments between recursive definitions at call sites and at the end of a function.

³ At the current stage, our prototype does not deal with loops.

Our implementation adapts from existing works [Chin et al. 2012; Qiu et al. 2013], which use a general strategy of unfolding a predicate in both the premise and conclusion until the entailment becomes obvious. [Chu et al. 2015] describes this strategy as “unfold-and-match” (U+M) and we will follow this terminology. To facilitate (U+M), we also follow [Qiu et al. 2013], unfolding a recursive constraint on a pointer x when its “footprint” (e.g., $x \rightarrow \text{next}$) is touched by the code.

Note that, in general, we would need to follow the rules in Fig. 2 and Fig. 3 to prove the relevant “evolution” and “enclosure” relations. However, for our key examples presented in Section 6.1 and Section 6.2, such relations trivially hold, thus we only discuss them briefly.

6.1 Summaries

In this first example, we consider the classic problem of copying a tree. Here the challenge is to prove that the copy, also a tree, is *isomorphic* to the original tree.

Consider the C-like program in Fig. 4. In Fig. 5 we define a standard binary tree and $\text{isocopy}(h_x, x, h_y, y)$ stating that the tree in h_x and rooted at x has a separate and isomorphic copy, housed in h_y and rooted at y .

```
struct node {
    int data;
    struct node *left, *right;
}

struct node *copytree(struct node *x) {
    if (!x) return 0;
    y = malloc(sizeof(struct node));
    y->data = x->data;
    y->left = copytree(x->left);
    y->right = copytree(x->right);
    return y;
}
```

Figure 4: Copy Tree Example

See the specification of the function copytree in Fig. 6. Note the conditions stated within the keyword **frame**. In copytree , updated cells are only those newly allocated, therefore, they are enclosed by the empty heap emp starting at the precondition. (Note that for all \mathcal{H} , $\text{emp} * \mathcal{H}$ holds.) Similarly, because \mathcal{H}_{ret} , the heap housing the newly produced copy of the original tree, contains only newly allocated cells, it is clear to see that it is evolved from an empty heap, thus $\text{EVOLVE}(\text{emp}, \mathcal{H}_{\text{ret}})$ holds.

Note that in the case that x is null, the proof is trivial by unfolding the definition of isocopy using the first rule. In Fig. 6, we consider the other case, i.e., $x \neq \text{null}$. We break down certain complex statements into simpler statements by introducing some temporary fresh variables. For readability, we simplify the formulas using variable substitutions and then eliminate redundant existential variables.

```
tree(h, x) :- h ≅ emp, x = null.
tree(h, x) :- h ≅ (x ↦ (·, left, right)) * hl * hr,
    tree(hl, left), tree(hr, right).

isocopy(hx, x, hy, y) :-
    hx ≅ emp, x = null, hy ≅ emp, y = null.
isocopy(hx, x, hy, y) :-
    hx ≅ (x ↦ (d, leftx, rightx)) * hxl * hxr,
    hy ≅ (y ↦ (d, lefty, righty)) * hyl * hyr, hx * hy,
    isocopy(hxl, leftx, hyl, lefty),
    isocopy(hxr, rightx, hyr, righty).
```

Figure 5: Definitions of tree and isocopy

We follow typical strategy in unfolding the recursive definition on x , i.e., $\text{tree}(\mathcal{H}_x, x)$, when the code touches x ’s “footprint” in program point 1. We proceed with program points 2, 3, and 4 by performing our symbolic execution together with some simplifications, mainly for readability. (We can see that at program point 4, the step is similar to our simple example discussed in Section 5 where a simple application of the frame rule allows us to maintain $\mathcal{H}_x \sqsubseteq \mathcal{M}$ while updating the data field of node y simultaneously.)

Let us now focus on some key steps in the proof. Importantly, at program point 5 and 6, where function calls necessitate compositional reasoning. We highlight the effects of our rules, namely (EV) and (FR).

Specifically, at the recursive call $\text{copytree}(1)$ (point 5), we need to prove that the assertion before this point implies the precondition of the function copytree , now would be:

$$\text{tree}(\mathcal{H}^l, l), \mathcal{H}^l \sqsubseteq \mathcal{M}.$$

Such proof can be achieved simply by matching \mathcal{H}^l with \mathcal{H}_1 . The assertion after this call (step 5) is then obtained by plugging the appropriate postcondition (local proof) and applying the frame rules. The proof in Fig. 6 are organized, hopefully in an intuitive way with explanatory text, to demonstrate how the rules work in combination. We use shaded boxes to further emphasize the effects of our rules (EV) and (FR). This explanation can be easily adapted for the recursive call at program point 6.

Finally, at program point 8, the postcondition is proved by renaming y to ret , a special variable denoting the returned value, and unfolding $\text{isocopy}(\mathcal{H}_x, x, \mathcal{H}_{\text{ret}}, \text{ret})$ in the postcondition using the second rule. We also need to perform other appropriate variable matching. Thus the proof used here is in fact an instance of the proof method named as U+M, mentioned above.

Remark: [Berdine et al. 2005b] used copytree program to demonstrate symbolic execution with Separation Logic, but proving a simpler property that the new tree produced is separate from the original one; they did not prove that the copy is *isomorphic* to the original tree. There have also been attempts in specifying isomorphic trees by introducing a mathematical tree τ in the SL assertion such as


```

requires:    tree( $\mathcal{H}_x, x$ ),  $\mathcal{H}_x \sqsubseteq \mathcal{M}$ 
ensures:    isocopy( $\mathcal{H}_x, x, \mathcal{H}_{ret}, ret$ ),  $\mathcal{H}_x \sqsubseteq \mathcal{M}$ ,  $\mathcal{H}_{ret} \sqsubseteq \mathcal{M}$ 
frame:      ENCLOSE(emp), EVOLVE(emp,  $\mathcal{H}_{ret}$ )
struct node *copytree(struct node *x) {
{ tree( $\mathcal{H}_x, x$ ),  $\mathcal{H}_x \sqsubseteq \mathcal{M}$  }
1  assume(x); l = x->left; r = x->right;
{  $\mathcal{H}_x \sqsubseteq (x \mapsto (-, l, r)) * \mathcal{H}_1 * \mathcal{H}_2$ , tree( $\mathcal{H}_1, l$ ), tree( $\mathcal{H}_2, r$ ),  $\mathcal{H}_x \sqsubseteq \mathcal{M}$  }
2  y = malloc(sizeof(struct node));
{  $\mathcal{H}_x \sqsubseteq (x \mapsto (-, l, r)) * \mathcal{H}_1 * \mathcal{H}_2$ , tree( $\mathcal{H}_1, l$ ), tree( $\mathcal{H}_2, r$ ), ( $y \mapsto (-, -, -)$ ) *  $\mathcal{H}_x \sqsubseteq \mathcal{M}$  }
3  d = x->data;
{  $\mathcal{H}_x \sqsubseteq (x \mapsto (d, l, r)) * \mathcal{H}_1 * \mathcal{H}_2$ , tree( $\mathcal{H}_1, l$ ), tree( $\mathcal{H}_2, r$ ), ( $y \mapsto (-, -, -)$ ) *  $\mathcal{H}_x \sqsubseteq \mathcal{M}$  }
4  y->data = d;
{  $\mathcal{H}_x \sqsubseteq (x \mapsto (d, l, r)) * \mathcal{H}_1 * \mathcal{H}_2$ , tree( $\mathcal{H}_1, l$ ), tree( $\mathcal{H}_2, r$ ), ( $y \mapsto (d, -, -)$ ) *  $\mathcal{H}_x \sqsubseteq \mathcal{M}$  }
  ↓ // (expanding the formula for better understanding)
  {  $\mathcal{H}_x \sqsubseteq (x \mapsto (d, l, r)) * \mathcal{H}_1 * \mathcal{H}_2$ , tree( $\mathcal{H}_1, l$ ), tree( $\mathcal{H}_2, r$ ), ( $y \mapsto (d, -, -)$ ) * ( $x \mapsto (d, l, r)$ ) *  $\mathcal{H}_1 * \mathcal{H}_2 \sqsubseteq \mathcal{M}$  }
5  z1 = copytree(l);
{ isocopy( $\mathcal{H}_1, l, \mathcal{H}'_1, z_1$ ),  $\mathcal{H}_1 \sqsubseteq \mathcal{M}$ ,  $\mathcal{H}'_1 \sqsubseteq \mathcal{M}$ ,           // postcondition of the call
   $\mathcal{H}_x \sqsubseteq (x \mapsto (d, l, r)) * \mathcal{H}_1 * \mathcal{H}_2$ , tree( $\mathcal{H}_2, r$ ),           // framed using (CFR)
  ( $y \mapsto (d, -, -)$ )  $\sqsubseteq \mathcal{M}$ , ( $x \mapsto (d, l, r)$ )  $\sqsubseteq \mathcal{M}$ ,  $\mathcal{H}_2 \sqsubseteq \mathcal{M}$ , // 'heap reality' framed through using (FR)
  ( $y \mapsto (d, -, -)$ ) * ( $x \mapsto (d, l, r)$ ) *  $\mathcal{H}_1 * \mathcal{H}_2 * \mathcal{H}'_1$  } // 'propagation of separation' using (EV)
6  z2 = copytree(r);
{ isocopy( $\mathcal{H}_2, r, \mathcal{H}'_2, z_2$ ),  $\mathcal{H}_2 \sqsubseteq \mathcal{M}$ ,  $\mathcal{H}'_2 \sqsubseteq \mathcal{M}$ ,           // postcondition of the call
  isocopy( $\mathcal{H}_1, l, \mathcal{H}'_1, z_1$ ),  $\mathcal{H}_x \sqsubseteq (x \mapsto (d, l, r)) * \mathcal{H}_1 * \mathcal{H}_2$ , // framed using (CFR)
   $\mathcal{H}_1 \sqsubseteq \mathcal{M}$ ,  $\mathcal{H}'_1 \sqsubseteq \mathcal{M}$ , ( $x \mapsto (d, l, r)$ )  $\sqsubseteq \mathcal{M}$ , ( $y \mapsto (d, -, -)$ )  $\sqsubseteq \mathcal{M}$ , // 'heap reality' framed through using (FR)
  ( $y \mapsto (d, -, -)$ ) * ( $x \mapsto (d, l, r)$ ) *  $\mathcal{H}_1 * \mathcal{H}_2 * \mathcal{H}'_1 * \mathcal{H}'_2$  } // 'propagation of separation' using (EV)
7  y->left = z1; y->right = z2;
{ isocopy( $\mathcal{H}_1, l, \mathcal{H}'_1, z_1$ ),  $\mathcal{H}_1 \sqsubseteq \mathcal{M}$ ,  $\mathcal{H}'_1 \sqsubseteq \mathcal{M}$ , isocopy( $\mathcal{H}_2, r, \mathcal{H}'_2, z_2$ ),  $\mathcal{H}_x \sqsubseteq (x \mapsto (d, l, r)) * \mathcal{H}_1 * \mathcal{H}_2$ ,  $\mathcal{H}_2 \sqsubseteq \mathcal{M}$ ,
   $\mathcal{H}'_2 \sqsubseteq \mathcal{M}$ , ( $x \mapsto (d, l, r)$ )  $\sqsubseteq \mathcal{M}$ , ( $y \mapsto (d, z_1, z_2)$ )  $\sqsubseteq \mathcal{M}$ , ( $y \mapsto (d, z_1, z_2)$ ) * ( $x \mapsto (d, l, r)$ ) *  $\mathcal{H}_1 * \mathcal{H}_2 * \mathcal{H}'_1 * \mathcal{H}'_2$  }
8  return y;
} { isocopy( $\mathcal{H}_x, x, \mathcal{H}_{ret}, ret$ ),  $\mathcal{H}_x \sqsubseteq \mathcal{M}$ ,  $\mathcal{H}_{ret} \sqsubseteq \mathcal{M}$  }

```

Figure 6: Proving Copy Tree Example

$tree(a, \tau) * tree(b, \tau)$ [O’Hearn 2012], meaning a and b are (roots of) isomorphic trees. We do not consider this as a satisfactory solution. Firstly, the disconnect of the mathematical tree from the program code makes it hard to automate the proof. Secondly, and more importantly, it is hardly possible to carry such assertion through subsequent code fragments to prove other properties. For example, after establishing $tree(a, \tau) * tree(b, \tau)$, we change the data at both cells a and b to a same value and finally want to prove that a and b are still isomorphic trees. We can imagine that automating such proof is quite challenging. However, we hope by now it is clear that, in our setting, it would be easy to prove the following triple:

```

{ isocopy( $\mathcal{H}_a, a, \mathcal{H}_b, b$ ),  $\mathcal{H}_a \sqsubseteq \mathcal{M}$ ,  $\mathcal{H}_b \sqsubseteq \mathcal{M}$  }
  assume(a);
  d = rand();
  a->data = d;
  b->data = d;
{ isocopy( $\mathcal{H}'_a, a, \mathcal{H}'_b, b$ ),  $\mathcal{H}'_a \sqsubseteq \mathcal{M}$ ,  $\mathcal{H}'_b \sqsubseteq \mathcal{M}$  }

```

```

struct node {
  int m;
  struct node *left, *right;
};
void mark(struct node *x) {
  if (!x || x->m == 1) return;
  struct node *l = x->left, *r = x->right;
  x->m = 1; mark(l); mark(r);
}

```

Figure 7: Graph Marking Algorithm

6.2 Structure Sharing

This example concerns *structure sharing* when dealing with graphs. Consider the classic C-like marking algorithm in Fig. 7. Initially the graph is unmarked, and we want to prove that at the end, the graph is fully marked. The first definition in Fig. 8 simply states that a graph is fully marked. A node is marked if its m field is 1; otherwise if the value is 0. Note that when `mark` is recursively invoked, because of sharing, what will be passed on is indeed a *partially marked graph*, as captured by the second definition in Fig. 8.

```

requires:    pmgraph( $\mathcal{H}_1, \mathcal{H}_2, x, t$ ),  $\mathcal{H}_1 \sqsubseteq \mathcal{M}$ ,  $\mathcal{H}_2 \sqsubseteq \mathcal{M}$ 
ensures:    mgraph( $\mathcal{H}'_1, \mathcal{H}'_2, x, t$ ),  $\mathcal{H}'_1 \sqsubseteq \mathcal{M}$ ,  $\mathcal{H}'_2 \sqsubseteq \mathcal{M}$ ,  $\text{dom}(\mathcal{H}_1) = \text{dom}(\mathcal{H}'_1)$ ,  $\text{dom}(\mathcal{H}_2) = \text{dom}(\mathcal{H}'_2)$ 
frame:      ENCLOSE( $\mathcal{H}_1 \cup \mathcal{H}_2$ ), EVOLVE( $\mathcal{H}_1, \mathcal{H}'_1$ ), EVOLVE( $\mathcal{H}_2, \mathcal{H}'_2$ )
void mark(struct node *x) {
{ pmgraph( $\mathcal{H}_1, \mathcal{H}_2, x, t$ ),  $\mathcal{H}_1 \sqsubseteq \mathcal{M}$ ,  $\mathcal{H}_2 \sqsubseteq \mathcal{M}$  }
1  assume( $x \&\& x \rightarrow m \neq 1$ ); l = x->left; r = x->right;
{  $\mathcal{H}_x \simeq (x \mapsto (0, l, r))$ ,  $x \notin t$ ,  $t_1 = t \cup \{x\}$ , pmgraph( $\mathcal{H}_{1a}, \mathcal{H}_{1b}, l, t_1$ ),  $\mathcal{H}_1 \simeq \mathcal{H}_x * \mathcal{H}_{1a} * \mathcal{H}_{1b}$ ,
   $t_2 = t_1 \cup \text{dom}(\mathcal{H}_{1a}) \cup \text{dom}(\mathcal{H}_{1b})$ , pmgraph( $\mathcal{H}_{2a}, \mathcal{H}_{2b}, r, t_2$ ),  $\mathcal{H}_2 \simeq \mathcal{H}_{2a} * \mathcal{H}_{2b}$ ,  $\mathcal{H}_1 * \mathcal{H}_2$ ,  $\mathcal{H}_1 \sqsubseteq \mathcal{M}$ ,  $\mathcal{H}_2 \sqsubseteq \mathcal{M}$  }
2  x->m = 1;
{ pmgraph( $\mathcal{H}_{1a}, \mathcal{H}_{1b}, l, t_1$ ),  $\mathcal{H}_{1a} * \mathcal{H}_{1b} * \mathcal{H}_{2a} * \mathcal{H}_{2b} * (x \mapsto (1, l, r)) \sqsubseteq \mathcal{M}$ ,
   $\mathcal{H}_1 \simeq \mathcal{H}_x * \mathcal{H}_{1a} * \mathcal{H}_{1b}$ ,  $\mathcal{H}_2 \simeq \mathcal{H}_{2a} * \mathcal{H}_{2b}$ ,
  pmgraph( $\mathcal{H}_{2a}, \mathcal{H}_{2b}, r, t_2$ ),  $x \notin t$ ,  $t_1 = t \cup \{x\}$ ,  $t_2 = t_1 \cup \text{dom}(\mathcal{H}_{1a}) \cup \text{dom}(\mathcal{H}_{1b})$  }
3  mark(l);
{ mgraph( $\mathcal{H}'_{1a}, \mathcal{H}'_{1b}, l, t_1$ ),  $\mathcal{H}'_{1a} \sqsubseteq \mathcal{M}$ ,  $\mathcal{H}'_{1b} \sqsubseteq \mathcal{M}$ ,  $\text{dom}(\mathcal{H}_{1a}) = \text{dom}(\mathcal{H}'_{1a})$ ,  $\text{dom}(\mathcal{H}_{1b}) = \text{dom}(\mathcal{H}'_{1b})$ , // post
   $\mathcal{H}_1 \simeq \mathcal{H}_x * \mathcal{H}_{1a} * \mathcal{H}_{1b}$ ,  $\mathcal{H}_2 \simeq \mathcal{H}_{2a} * \mathcal{H}_{2b}$ , // (CFR)
  pmgraph( $\mathcal{H}_{2a}, \mathcal{H}_{2b}, r, t_2$ ),  $x \notin t$ ,  $t_1 = t \cup \{x\}$ ,  $t_2 = t_1 \cup \text{dom}(\mathcal{H}_{1a}) \cup \text{dom}(\mathcal{H}_{1b})$ , // (CFR)
   $\mathcal{H}'_{1a} * \mathcal{H}_{1b} * \mathcal{H}_{2a} * \mathcal{H}_{2b} * (x \mapsto (1, l, r))$ ,  $\mathcal{H}'_{1b} * \mathcal{H}_{1a} * \mathcal{H}_{2a} * \mathcal{H}_{2b} * (x \mapsto (1, l, r))$ , // (EV)
   $\mathcal{H}_{2a} * \mathcal{H}_{2b} * (x \mapsto (1, l, r)) \sqsubseteq \mathcal{M}$  } // (FR)
4  mark(r);
{ mgraph( $\mathcal{H}'_{1a}, \mathcal{H}'_{1b}, l, t_1$ ),  $\text{dom}(\mathcal{H}_{1a}) = \text{dom}(\mathcal{H}'_{1a})$ ,  $\text{dom}(\mathcal{H}_{1b}) = \text{dom}(\mathcal{H}'_{1b})$  // (CFR)
   $\mathcal{H}_1 \simeq \mathcal{H}_x * \mathcal{H}_{1a} * \mathcal{H}_{1b}$ ,  $\mathcal{H}_2 \simeq \mathcal{H}_{2a} * \mathcal{H}_{2b}$ , // (CFR)
   $x \notin t$ ,  $t_1 = t \cup \{x\}$ ,  $t_2 = t_1 \cup \text{dom}(\mathcal{H}_{1a}) \cup \text{dom}(\mathcal{H}_{1b})$ , // (CFR)
  mgraph( $\mathcal{H}'_{2a}, \mathcal{H}'_{2b}, r, t_2$ ),  $\mathcal{H}'_{2a} \sqsubseteq \mathcal{M}$ ,  $\mathcal{H}'_{2b} \sqsubseteq \mathcal{M}$ ,  $\text{dom}(\mathcal{H}_{2a}) = \text{dom}(\mathcal{H}'_{2a})$ ,  $\text{dom}(\mathcal{H}_{2b}) = \text{dom}(\mathcal{H}'_{2b})$  // post
   $\mathcal{H}'_{2a} * \mathcal{H}'_{1a} * \mathcal{H}_{1b} * \mathcal{H}_{2b} * (x \mapsto (1, l, r))$ ,  $\mathcal{H}'_{2b} * \mathcal{H}'_{1b} * \mathcal{H}_{1a} * \mathcal{H}_{2a} * (x \mapsto (1, l, r))$ , // (EV)
   $\mathcal{H}'_{1a} \sqsubseteq \mathcal{M}$ ,  $\mathcal{H}'_{1b} \sqsubseteq \mathcal{M}$ ,  $(x \mapsto (1, l, r)) \sqsubseteq \mathcal{M}$  } // (FR)
} { mgraph( $\mathcal{H}'_1, \mathcal{H}'_2, x, t$ ),  $\mathcal{H}'_1 \sqsubseteq \mathcal{M}$ ,  $\mathcal{H}'_2 \sqsubseteq \mathcal{M}$ ,  $\text{dom}(\mathcal{H}_1) = \text{dom}(\mathcal{H}'_1)$ ,  $\text{dom}(\mathcal{H}_2) = \text{dom}(\mathcal{H}'_2)$  }

```

Figure 9: Mark Graph Example

```

mgraph( $h_1, h_2, x, t$ ) :-  $h_1 \simeq \text{emp}$ ,  $h_2 \simeq \text{emp}$ ,  $x = \text{null}$ .
mgraph( $h_1, h_2, x, t$ ) :-  $h_1 \simeq (x \mapsto (1, \text{left}, \text{right}))$ ,
   $x \in t$ ,  $h_2 \simeq \text{emp}$ , mgraph( $-, -, x, \emptyset$ ).
mgraph( $h_1, h_2, x, t$ ) :-  $h_x \simeq (x \mapsto (1, \text{left}, \text{right}))$ ,
   $x \notin t$ ,  $t_1 = t \cup \{x\}$ , mgraph( $h_{1a}, h_{1b}, \text{left}, t_1$ ),
   $h_1 \simeq h_x * h_{1a} * h_{1b}$ ,  $t_2 = t_1 \cup \text{dom}(h_{1a}) \cup \text{dom}(h_{1b})$ ,
  mgraph( $h_{2a}, h_{2b}, \text{right}, t_2$ ),  $h_2 \simeq h_{2a} * h_{2b}$ ,  $h_1 * h_2$ .

pmgraph( $h_1, h_2, x, t$ ) :-  $h_1 \simeq \text{emp}$ ,  $h_2 \simeq \text{emp}$ ,  $x = \text{null}$ .
pmgraph( $h_1, h_2, x, t$ ) :-  $h_1 \simeq (x \mapsto (1, \text{left}, \text{right}))$ ,
   $x \in t$ ,  $h_2 \simeq \text{emp}$ , mgraph( $-, -, x, \emptyset$ ).
pmgraph( $h_1, h_2, x, t$ ) :-  $h_x \simeq (x \mapsto (0, \text{left}, \text{right}))$ ,
   $x \notin t$ ,  $t_1 = t \cup \{x\}$ , pmgraph( $h_{1a}, h_{1b}, \text{left}, t_1$ ),
   $h_1 \simeq h_x * h_{1a} * h_{1b}$ ,  $t_2 = t_1 \cup \text{dom}(h_{1a}) \cup \text{dom}(h_{1b})$ ,
  pmgraph( $h_{2a}, h_{2b}, \text{right}, t_2$ ),  $h_2 \simeq h_{2a} * h_{2b}$ ,  $h_1 * h_2$ .

```

Figure 8: Definitions of mgraph and pmgraph

In this algorithm, the programmer has made a *crucial* assumption that if a node has been marked, all nodes reachable from it are marked. This *invariant* property allows us to stop when an already marked node is encountered. Consequently, if we pass in as input a graph with some randomly marked nodes, this algorithm will not work. We note further that both pmgraph and graph definitions reflect this assumption.

In Fig. 9 we show the specification of the function mark and the proof for the most interesting case: x is not null and

its m field has not been marked. The program does not deallocate any memory cells, thus $\text{dom}(\mathcal{H}_1) = \text{dom}(\mathcal{H}'_1)$ implies $\text{EVOLVE}(\mathcal{H}_1, \mathcal{H}'_1)$ and $\text{dom}(\mathcal{H}_2) = \text{dom}(\mathcal{H}'_2)$ implies $\text{EVOLVE}(\mathcal{H}_2, \mathcal{H}'_2)$. In other words, we do not need to separately construct a proof for the evolution relation.

The assertion after step 1 is obtained, as before, by unfolding the definition of pmgraph using the third rule and instantiating the values of l and r . Note that this unfolding is triggered since the footprint of x is touched. (Using the other rules will lead to a conflict with the constraint $\text{assume}(x \&\& x \rightarrow m \neq 1)$.) At the recursive call mark(1) (point 3), we need to prove that the assertion after program point 2 implies the precondition of the function mark, now would be:

$$\text{pmgraph}(\mathcal{H}_1^l, \mathcal{H}_2^l, l, t^l), \mathcal{H}_1^l \sqsubseteq \mathcal{M}, \mathcal{H}_2^l \sqsubseteq \mathcal{M}.$$

Such proof can be achieved simply by matching \mathcal{H}_1^l with \mathcal{H}_{1a} , \mathcal{H}_2^l with \mathcal{H}_{1b} , and t^l with t_1 .

The assertion after this call (step 3) is then obtained by application of framing. First we use the specification to replace one occurrence of pmgraph (the first one) by mgraph; note that $\mathcal{H}_{1a} \sqsubseteq \mathcal{M}$ and $\mathcal{H}_{1b} \sqsubseteq \mathcal{M}$ are removed. What we would like to focus on here is the shaded heap formula, which was framed through step 3 because this heap lies outside the updates of the recursive call mark(1). For this step, $\mathcal{H}_{1a}, \mathcal{H}_{1b}$ evolve into $\mathcal{H}'_{1a}, \mathcal{H}'_{1b}$, so a heap's separation from $\mathcal{H}_{1a}, \mathcal{H}_{1b}$ before the step was propagated into its separa-

tion from $\mathcal{H}'_{1a}, \mathcal{H}'_{1b}$ after the step. This explanation is easily adapted for the call at program point 4. The postcondition is proved by unfolding $\text{mgraph}(\mathcal{H}'_1, \mathcal{H}'_2, x, t)$ using the third rule, followed by appropriate variable matching.

Remark: One might argue that our top-level specification mgraph is contrived so as to be similar to pmgraph (which necessarily reflects the traversal order of the graph), and prefer another definition of a (fully) marked graph, for example, by defining that graph as a collection of edges, and that all its connecting vertices are marked. This definition does not reflect the above mentioned assumption of the programmer, thus it is not appropriate to immediately act as the postcondition for the recursive function mark . Instead, we could attempt to connect mgraph , where t is instantiated as an empty set, to the new definition. Ultimately, it is about finally connecting the specifications of the code with some “declarative” specifications that are independent of any code. We stress here that this final step is a theorem-proving issue, which arises frequently in practice, e.g., when verifying sorting algorithms. Such issue is beyond the scope of this paper.

7. Related Work

It is possible, but very difficult, to reason in Hoare logic about programs which modify data structures defined by pointers. [Morris 1982; Bornat 2000] explore this direction. The resulting proofs are rather inelegant and remain too low-level to be widely applicable, let alone being automated.

Separation Logic (SL) [O’Hearn et al. 2001; Reynolds 2002] was a significant advance with local reasoning via a frame rule, influencing modern verification tools. For example, [Berdine et al. 2005a; Botinčan et al. 2009; Jacobs et al. 2011] implement SL-based symbolic execution, as described in [Berdine et al. 2005b]. But there was a problem in accommodating data structures with *unrestricted sharing*.

[Bornat et al. 2004] present a pioneering SL-based approach for reasoning about data structures with intrinsic sharing. The attempt results in “dauntingly subtle” [Bornat et al. 2004] definitions and verifications. Thus it is unclear how to automate such proofs.

Explicit naming of heaps naturally emerged as extensions of SL [Duck et al. 2013]. [Reynolds 2003] conjectured that referring explicitly to the current heap in specifications would allow better handles on data structures with sharing. It is our current work that realizes the conjecture by connecting the explicit subheaps to the current global heap (\mathcal{M}) and formalizing the concepts of “evolution” and “enclosure”. This leads to a new frame rule, and consequently gains back the power of compositional reasoning.

Next consider [Hobor and Villard 2013] which addressed sharing (but does not address automation). Recall the mgraph function, but now consider its application on a DAG, Fig. 10. The “ramify” rule in [Hobor and Villard 2013] would at-

tempt to isolate the shaded heap portion 1 and prove that the portion 1 has all been marked. With the help of the *magic wand*, this seems general enough. Its application, however, is counter-intuitive and hard to automate, because the portion 1 is *artificial*: it does not correspond to the actual traversal of the code.

In somewhat related thread of work, [Nanevski et al. 2010] show that by choosing less straightforward definitions of heaps and of heap union in Coq, we can obtain effective reasoning in the presence of abstract heap variables, and hence support full separation assertion logic without resulting in excessive proof obligations. As a result, proofs of a number of simple but realistic programs have been successfully mechanized, though not automated.

There are other works related to concept of explicit heaps with framing. Examples are (Implicit) Dynamic Frames [Kassios 2006; Smans et al. 2009] and Region Logic [Banerjee et al. 2008, 2013]. The underlying approach is to represent the heap \mathcal{M} as a (possibly implicit) *total map* over all possible addresses, and to represent access or modification rights as sets of addresses F . Separation is represented as set disjointness, i.e., $F_1 \cap F_2 = \emptyset$. In [Banerjee et al. 2013], a new form of frame condition specifies write, read, and allocation effects using region expressions; this supports a “frame rule” that allows a command to read state on which the framed predicate depends.

While our work shares the same underlying principles on which the concept of framing operates, there are important differences. Firstly, our frames are of type heap (as opposed to just sets). On one hand, they are constrained by the accompanied recursive predicates; on the other hand, they are strongly connected to the program semantics via the heap reality constraints, e.g., $\mathcal{H}_i \sqsubseteq \mathcal{M}$. Secondly, our frames can be packaged under a program verification framework where: (a) with *symbolic execution*, frame information is propagated precisely through straight-line fragments to abstraction boundary points; and (b) with a new *frame rule*, “separation” of frames are propagated automatically through the abstraction boundary points for *successive* uses of framing, as in the case of the mark graph example. Furthermore, we can identify a good approximation of the conditions needed for framing, i.e., rules in Figures 2 and 3, which are effectively checkable. In the end, current works on dynamic frames and regions do not have the expressiveness that allow automatic program verification to our level, as evidenced by our driving examples in Section 6.

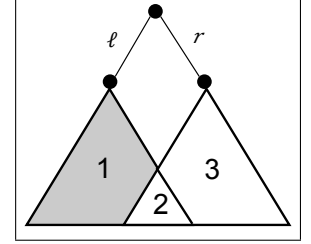


Figure 10: mark DAG

8. Conclusion

We started with the program logic which extends Separation Logic by allowing the use of explicit subheaps in assertions. The axiomatization of this logic was based on strongest post-conditions; thus the logic is *automatable* by symbolic execution armed with a suitable constraint solver. In this paper, we developed a new frame rule for this logic and thus raised the level of automation to be *compositional*. The two key concepts were: *evolution*, which essentially captures the enlargement of a heap; and *enclosure*, which encircles heap updates into prescribed areas. Combining these two allows for framing properties of heaps away from program updates. We successfully applied this rule to two breakthrough examples, demonstrating its potential for a new level of automatic verification of data structures.

References

- A. Banerjee, D. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, pages 387–411, 2008.
- A. Banerjee, D. A. Naumann, and S. Rosenberg. Local reasoning for global invariants, part i: Region logic. *J. ACM*, 60(3):18:1–18:56, 2013.
- J. Berdine, C. Calcagno, and P. O’Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCQ*, pages 115–137, 2005a.
- J. Berdine, C. Calcagno, and P. O’Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005b.
- R. Bornat. Proving pointer programs in hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.
- R. Bornat, C. Calcagno, and P. O’Hearn. Local reasoning, separation, and aliasing. In *Proc. 2nd workshop on Semantics, Program Analysis and Computing Environments for Memory Management*, 2004.
- M. Botinčan, M. Parkinson, and W. Schulte. Separation logic verification of C programs with an SMT solver. *Electronic Notes in Theoretical Computer Science*, 254:5–23, Oct. 2009.
- W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. In *Science of Computer Programming*, 77(9), pages 1006–1036, 2012.
- D. H. Chu, J. Jaffar, and M. T. Trinh. Automatic induction proofs of data-structures in imperative programs. In *PLDI*, pages 457–466, 2015.
- G. Duck, J. Jaffar, and N. Koh. Constraint-based program reasoning with heaps and separation. In *CP*, pages 282–298, 2013.
- C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 1969.
- A. Hobor and J. Villard. The ramifications of sharing in data structures. In *POPL*, pages 523–536, 2013.
- B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NFM*, pages 41–55, 2011.
- J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *POPL*, pages 111–119, 1987.
- I. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, pages 268–283, 2006.
- J. M. Morris. A general axiom of assignment. assignment and linked data structures. a proof of the schorr-waite algorithm. In *Theoretical Foundations of Programming Methodology*, 1982.
- A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *Symposium on Principles of Programming Languages*, January 2010.
- P. O’Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, pages 1–19, 2001.
- P. W. O’Hearn. A primer on separation logic (and automatic program verification and analysis). In *Software Safety and Security - Tools for Analysis and Verification*, pages 286–318. IOS Press, 2012.
- X. K. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI*, pages 231–242, 2013.
- J. C. Reynolds. Separation logic: A logic for shared mutable data objects. In *LICS*, pages 55–74, 2002.
- J. C. Reynolds. A short course on separation logic. <http://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr/wwwa2003/aac.html>, 2003.
- J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, pages 148–172, 2009.